SAND2015-9205R

Parallel Tensor Compression for Large-Scale Scientific Data

Woody Austin
University of Texas, Austin, TX, USA
Email: woody.n.austin@gmail.com

Grey Ballard and Tamara G. Kolda
Sandia Nationals Labs, Livermore, CA, USA
Email: gmballa@sandia.gov, tgkolda@sandia.gov

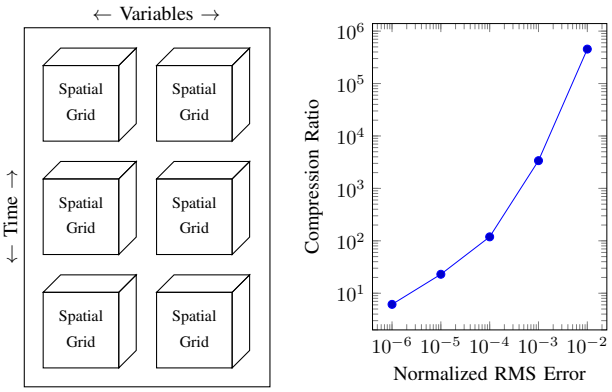
***Abstract*—As parallel computing trends towards the exascale, scientific data produced by high-fidelity simulations are growing increasingly massive. For instance, a simulation on a three-dimensional spatial grid with 512 points per dimension that tracks 64 variables per grid point for 128 time steps yields 8 TB of data. By viewing the data as a dense five-way tensor, we can compute a Tucker decomposition to find inherent low-dimensional multilinear structure, achieving compression ratios of up to 10000 on real-world data sets with negligible loss in accuracy. So that we can operate on such massive data, we present the first-ever distributed-memory parallel implementation for the Tucker decomposition, whose key computations correspond to parallel linear algebra operations, albeit with nonstandard data layouts. Our approach specifies a data distribution for tensors that avoids any tensor data redistribution, either locally or in parallel. We provide accompanying analysis of the computation and communication costs of the algorithms. To demonstrate the compression and accuracy of the method, we apply our approach to real-world data sets from combustion science simulations. We also provide detailed performance results, including parallel performance in both weak and strong scaling experiments.**

***Keywords*-Tucker tensor decomposition; compression**

I. Introduction

Today's high-performance parallel computers enable large-scale, high-fidelity simulations of natural phenomena across scientific domains. As the speed and quality of simulations increases, the amount of data produced is growing at a rate that is creating bottlenecks in the scientific process. A posteriori analysis of the data requires dedicated storage devices and parallel clusters even for simple computations. One of the primary goals of this work is to supply a compression technique for large-scale simulation data, enabling much more efficient data storage, transfer, and analysis and facilitating bigger and better science.

Scientific simulation data is naturally multidimensional, tracking different variables in space and time; see Fig. 1a. As a prototypical example, we focus on simulation data from combustion science research. In this domain, simulated phenomena tend to be bursty, with important activity occurring in subsets of the spatial grid, small points in time, or involving a subset of the quantities of interest, like chemical species or fluid velocities. Thus, the data typically have low-dimensional multilinear structure allowing for compression. We consider compression based on the Tucker decomposition for higher-order tensors, which is analogous to principal component analysis (PCA) or the truncated singular value decomposition (T-SVD) of two-way data. We describe the Tucker decomposition in more detail in Sec. II.



(a) Natural five-way multiway structure of scientific data.

(b) Compression vs. fidelity for 512 GB simulation dataset.

Figure 1.

Our main contribution in this work is a distributed-memory parallel algorithm and implementation for computing the Tucker decomposition of general dense tensors. Compression rates for a 512 GB scientific simulation dataset using our method are shown in Fig. 1b. To the best of our knowledge, ours is the first distributed-memory implementation of a parallel algorithm for computing a Tucker decomposition. Related work for other decompositions and algorithmic kernels are discussed in Sec. III.

The algorithm works for dense tensors of any order (i.e., number of dimensions) and size, given adequate memory, e.g., three times the size of the data. The algorithm is efficient because it casts local computations in terms of BLAS3 routines to exploit optimized, architecture-specific kernels; the data distributions and corresponding parallel computations are designed to reduce interprocessor communication. We present the data distribution, parallel kernels, and overall algorithm in Secs. IV to VI, along with accompanying analysis of the computation and communication costs and memory requirements.

Using real-world simulation data from combustion science, we demonstrate the effectiveness of Tucker for compression in Sec. VII. In particular, we show that these data sets have inherent low-dimensional multilinear structure that

can be exploited for compression. We show that the Tucker tensor decomposition can reduce the data by 50–99.98% with normalized root mean squared (RMS) errors less than 10^{-6}, and by 99.9% and more with normalized RMS errors less than 10^{-2}. Such high compression rates allow terabytes of data to be reduced to gigabytes or megabytes, enabling easy data transfer and sharing. Additionally, we can reconstruct small subsets of the data upon request, enabling efficient analysis on even a single laptop.

The implementation is scalable. Results in Sec. VIII show that the algorithm performs well for large and small data sets using up to 30,000 cores. It achieves near peak performance, as high as 83%, on a single node consisting of 24 cores and up to 17% of peak on over 1000 nodes. At large scale, we are able to compress a 15 TB data set to 1.5 GB in about a minute and a 9 GB data set to 33 MB in under a second, with aggregate performance up to 100 TFLOPS.

II. Tucker Tensor Decomposition

A. Tensor Notation and Operations

Let $\mathcal{X}$ be a real-valued tensor of size $I_1 \times I_2 \times \cdots \times I_N$. We define

$$I = \prod_{n=1}^{N} I_n \quad \text{and} \quad \hat{I}_n = I/I_n \text{ for } n \in \{1, \ldots, N\}$$

to be the total number of data elements and that number divided by the length of mode n, respectively. The *mode-n unfolding* rearranges the elements of $\mathcal{X}$ to form an $I_n \times \hat{I}_n$ matrix and is denoted by $\mathbf{X}_{(n)}$. Tensor element $(i_1, i_2, \ldots, i_N)$ maps to matrix element (i_n, j) where $j = 1 + \sum_{k=1}^{n-1}(i_k - 1)\hat{I}_k + \sum_{k=n+1}^{N}(i_k - 1)(\hat{I}_k/I_n)$. The *$n$-rank* of the tensor $\mathcal{X}$ is the column rank of $\mathbf{X}_{(n)}$, denoted $\text{rank}_n(\mathcal{X})$. The *norm* of a tensor is the square root of the sum of the squares of the entries, i.e., $\|\mathcal{X}\| = \|\mathbf{X}_{(1)}\|_F$.

The *mode-n product* of the tensor $\mathcal{X}$ with a real-valued matrix $\mathbf{V}$ of size $I_n \times J$ is denoted $\mathcal{X} \times_n \mathbf{V}$ and is of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$. This can be expressed in terms of unfolded tensors, i.e.,

$$\mathcal{Y} = \mathcal{X} \times_n \mathbf{V} \quad \Leftrightarrow \quad \mathbf{Y}_{(n)} = \mathbf{V}\mathbf{X}_{(n)}.$$

This is also known as the tensor-times-matrix (TTM) product. The order of multiplications is irrelevant, i.e., $\mathcal{X} \times_m \mathbf{W} \times_n \mathbf{V} = \mathcal{X} \times_n \mathbf{V} \times_m \mathbf{W}$ for $m \neq n$. If we are multiplying by a sequence of matrices, then we may use the shorthand $\mathcal{Y} = \mathcal{X} \times \{\mathbf{V}^{(n)}\}$, to indicate that we should multiply $\mathcal{X}$ by each matrix in the set along the corresponding mode. Unless otherwise indicated, we assume $\{\mathbf{V}^{(n)}\}$ is indexed from $n = 1, \ldots, N$.

B. Tucker Algorithm and Key Kernels

The Tucker decomposition [20] approximates a data tensor $\mathcal{X}$ as

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{U}^{(1)} \times_2 \mathbf{U}^{(2)} \cdots \times_N \mathbf{U}^{(N)} = \mathcal{G} \times \{\mathbf{U}^{(n)}\},$$

where $\mathcal{G}$ is the *core tensor* of size $R_1 \times R_2 \times \cdots \times R_N$ and $\mathbf{U}^{(n)}$ is a *factor matrix* of size $I_n \times R_n$ for $n = 1, \ldots, N$. We say $R_1 \times R_2 \times \cdots \times R_N$ is the reduced dimension or, equivalently, the rank of the reduced representation. The decomposition is illustrated for $N = 3$ in Fig. 2.

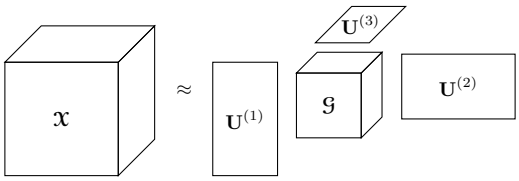


Figure 2. Tucker decomposition for $N = 3$.

Ideally, the data tensor has low-rank structure meaning that we can choose $R_n \approx \text{rank}_n(\mathcal{X}) \ll I_n$ so that most of the variance of the data is preserved. Given factor matrices $\{\mathbf{U}^{(n)}\}$, it is well known that the optimal core is given by $\mathcal{G} = \mathcal{X} \times \{\mathbf{U}^{(n)}\}$ [13].

The storage is dominated by the core of size $R = \prod_{n=1}^{N} R_n$. There is additional $\sum_{n=1}^{N} I_n R_n$ storage for the factor matrices, but this is generally negligible compared to the storage of the core.

The Tucker1 method, better known as truncated higher-order singular value decomposition (T-HOSVD) [6], [20], is a particular case of Tucker where $\mathbf{U}^{(n)}$ is set to be the R_n leading left singular vectors of $\mathbf{X}_{(n)}$. The T-HOSVD is not optimal, but it is often a good starting point for the iterative procedure described below. We use a variation known as the *sequentially-truncated HOSVD* (ST-HOSVD) [21] for initialization. The first factor matrix is initialized as for the T-HOSVD, i.e., the R_1 leading left singular values of $\mathbf{X}_{(1)}$. The nth factor matrix is initialized as the R_n leading left singular vectors of $\mathbf{Y}_{(n)}$ where $\mathcal{Y} = \mathcal{X} \times_1 \mathbf{U}^{(1)\mathsf{T}} \cdots \times_{n-1} \mathbf{U}^{(n-1)\mathsf{T}}$. The size of $\mathbf{Y}_{(n)}$ is $I_n \times (\prod_{m<n} R_m)(\prod_{m>n} I_m)$. One advantage of this method is that the $\mathcal{Y}$ tensors are smaller than $\mathcal{X}$ for $n > 1$. The ST-HOSVD is presented in Alg. 1. Here, we pick the R_n values according to a user-specified relative error threshold [21]. Although we do not make it explicit in the algorithm, the modes can be processed in any arbitrary order; see Sec. VIII-C for the impact of different orderings.

Algorithm 1 Sequentially-Truncated HOSVD (ST-HOSVD)

```
1:  procedure ST-HOSVD(𝒳, ε)
2:      𝒴 ← 𝒳
3:      for n = 1, ..., N do
4:          S ← Y_(n) Y_(n)^T
5:          R_n ← min R such that Σ_{r>R} λ_r(S) ≤ ε²‖𝒳‖²/N
6:          U^(n) ← leading R_n eigenvectors of S
7:          𝒴 ← 𝒴 ×_n U^(n)T
8:      end for
9:      𝒢 ← 𝒴
10:     return (𝒢, {U^(n)})
11: end procedure
```

The higher-order orthogonal iteration (HOOI) [7], [14] is an alternating optimization method that further improves the approximation. The procedure cycles through the modes of the tensor, calculating the leading left singular vectors of $\mathbf{Y}_{(n)}$ where $\mathcal{Y} = \mathcal{X} \times \{ \mathbf{U}^{(m)\mathsf{T}} \}_{m \neq n}$, i.e., $\mathcal{X}$ is multiplied in every mode except n by the corresponding factor matrix. The size of $\mathbf{Y}_{(n)}$ is $I_n \times \hat{R}_n$ where $\hat{R}_n = \prod_{m \neq n} R_m$. The HOOI method is presented in Alg. 2. HOOI is an iterative algorithm that monotonically improves the error but has no guarantees on converging to a global minimum; we iterate until the approximation error is small enough, the improvement in approximation falls below a given threshold, or a maximum number of iterations are reached. We track the quantity $(\|\mathcal{X}\|^2 - \|\mathcal{G}\|^2)$ in line 10 because it is equivalent to the fit of the model, i.e., $\|\mathcal{X} - \mathcal{G} \times \{ \mathbf{U}^{(n)} \}\|^2$ [13].

Algorithm 2 Higher-order Orthogonal Iteration (HOOI)

```
1:  procedure HOOI(X, ε)
2:      (G, {U^(n)}) = ST-HOSVD(X, ε)
3:      repeat
4:          for n = 1, ..., N do
5:              Y ← X × {U^(m)T}_{m≠n}
6:              S ← Y_(n) Y_(n)^T
7:              U^(n) ← leading R_n eigenvectors of S
8:          end for
9:          G ← Y ×_N U^(N)T
10:     until the quantity (||X||^2 − ||G||^2) ceases to decrease
11:     return (G, {U^(n)})
12: end procedure
```

For both ST-HOSVD and HOOI, we compute the leading left singular vectors of $\mathcal{Y}$ by forming its $I_n \times I_n$ Gram matrix $\mathbf{S} = \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^{\mathsf{T}}$ and then computing its eigenvectors. Alternatively, we could work directly with $\mathbf{Y}_{(n)}$ and compute its singular vectors (see Sec. IX for more discussion). Our decision is motivated by the application-based assumption that ϵ is larger than the square root of machine precision and that I_n is relatively small, i.e., $I_n \leq 2000$ for all n.

We focus on parallelizing the ST-HOSVD and HOOI methods whose inputs are $\mathcal{X}$ and the desired accuracy. The three key operations in Alg. 1 and Alg. 2 are

1) the (sequence of) TTM operations to calculate $\mathcal{Y}$,
2) the Gram matrix computations to calculate $\mathbf{S}$, and
3) the eigenvector calculations to calculate $\mathbf{U}^{(n)}$.

The difference between the two algorithms is that the tensors $\mathcal{Y}$ are of different sizes. In HOOI, the core matrix is computed in line 9, exploiting the fact that the current $\mathcal{Y}$ tensor already has the first $N-1$ products calculated.

C. Reconstruction

Given a core $\mathcal{G}$ and factor matrices $\{ \mathbf{U}^{(n)} \}$, we compute the approximate reconstruction of $\mathcal{X}$, denoted $\tilde{\mathcal{X}}$, by calling a sequence of TTM operations, i.e.,

$$\mathcal{X} \approx \tilde{\mathcal{X}} = \mathcal{G} \times \{ \mathbf{U}^{(n)} \}. \tag{1}$$

Note that we can efficiently compute subtensors of $\tilde{\mathcal{X}}$ (without forming the entire tensor) by modifying eq. (1) appropriately, using subsets of rows of the factor matrices.

III. Related Work

The Tucker decomposition is a powerful tool for compression of scientific data, as previously shown for hyperspectral images [10] and volume rendering [1]. Our work parallelizes the method and demonstrates its utility on large-scale data sets.

To the best of our knowledge, ours is the first distributed memory implementation of the Tucker tensor decomposition. Zhou, Cichocki, and Xie [23] propose a randomized method for computing the Tucker decomposition of large tensors, but still assume all the data fits on a single machine. Li et al. [15] consider a shared-memory parallel implementation of dense TTM, which may be used directly and adapted for the local computations in our method.

Several groups have parallelized the canonical polyadic (CP) tensor decomposition. The primary kernels are distinct from this work, but we mention a few similarities. Karlsson, Kressner, and Uschmajew [11] have a parallel CP decomposition of multidimensional data with missing entries using either cyclic coordinate descent or alternating least squares; they use a similar data distribution strategy to our in terms of the data tensor and factor matrices. Kaya and Uçar [12] focus on CP of *sparse* tensors using a hypergraph partitioning scheme. Kang et al. [9] provide a MapReduce implementation of CP for sparse data. Smith et al. [18] parallelize a particular key kernel for CP on sparse data for three-mode tensors. Phan and Cichocki [16] break up the problem by computing CP decompositions of subtensors (in parallel) and then stitching the results together for a global CP decomposition.

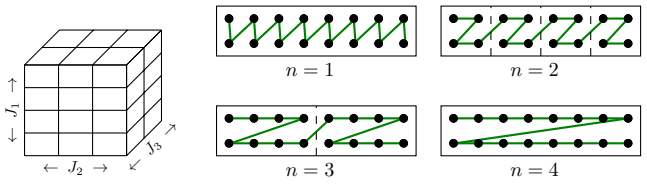
Another tensor decomposition known as the tensor train (TT) decomposition has recently been parallelized by Etter [8] by focusing on the recursive branching nature of the method. There has also been considerable work on parallel tensor contraction; see, e.g., [17] and references therein.

IV. Parallel Data Distributions

For N-way tensors, we assume a logical N-way processor grid. Let $P_1 \times P_2 \times \cdots \times P_N$ be the size of the processor grid. For ease of presentation, we assume P_n evenly divides I_n and R_n, but our implementation does not require this. Let $P = \prod P_n$ be the total number of processors and $\hat{P}_n = P/P_n$ be the number of processors in all modes but n. From the point of view of a particular processor, we denote its local portion of a distributed object with an overhead bar.

A. Tensor Distribution

Let $J_1 \times J_2 \times \cdots \times J_N$ be the size of a generic tensor $\mathcal{Y}$ where, for our purposes, $J_n \in \{ I_n, R_n \}$ for all n. Let $J = \prod J_n$ be the total size and $\hat{J}_n = J/J_n$ be the size in



(a) Tensor on $4\times3\times2$ processor grid.

(b) Local data layout for $2\times2\times2\times2$ tensor unfolded in different modes.

Figure 3. Tensor data distribution/layout examples.

all modes but n. We impose a Cartesian parallel distribution of the tensor across processors, which we refer to as a *block distribution*. Each processor owns a distinct subtensor of size $J_1/P_1 \times \cdots \times J_N/P_N$, with J/P entries. A similar tensor distribution is used in [23]. See Fig. 3a for an illustration of a 3D tensor block distributed over a $4\times3\times2$ processor grid. The local tensor, $\bar{\mathcal{Y}}$, is stored so that its mode-1 unfolding is in column-major order.

B. Matrix Distribution

Factor matrices are also distributed across processors. Given a mode n, let $\mathbf{V}$ be a generic matrix of size $K \times J$ where $(K, J) = (R_n, I_n)$ for decomposition or $(K, J) = (I_n, R_n)$ for reconstruction. We treat our processor grid as two-dimensional of size $P_n \times \hat{P}_n$. We divide $\mathbf{V}$ into P_n *column blocks* so that $\mathbf{V} = \begin{bmatrix}\mathbf{V}_1 & \mathbf{V}_2 & \cdots & \mathbf{V}_{P_n}\end{bmatrix}$. We distribute the matrix *redundantly* on every processor column, i.e., $\hat{P}_n$ times. Since the these matrices are relatively small, the redundant storage is negligible. More precisely, if $R_n < \hat{I}_n/\hat{P}_n$, then the local storage of a factor matrix does not exceed the size of the local tensor. Processor $(p_1, p_2, \ldots, p_N)$ owns block column $\mathbf{V}_{p_n}$ of size $K \times J/P_n$. We assume the local matrices are stored in row-major order.

C. Unfolded Tensor Distribution

Unfolding a tensor is a purely logical process and involves no data redistribution. Given a block distribution of a tensor, the unfolded tensor (a matrix) is also block distributed across a 2D processor grid. In other words, a tensor unfolded in mode n has dimension $J_n \times \hat{J}_n$ and is distributed over a $P_n \times \hat{P}_n$ processor grid. Note that if $P_n = 1$, then the unfolded matrix has a 1D column distribution across P processors.

The local portion of the unfolded tensor is equivalent to unfolding the local tensor. Again, this unfolding is logical; no local data distribution is required. The local unfolded tensor, $\bar{\mathbf{Y}}_{(n)}$, is stored in $\prod_{m>n}(I_m/P_m)$ block columns, with each block column of size $(I_n/P_n) \times \prod_{m<n}(I_m/P_m)$ stored in row-major order. The local data layout is illustrated in Fig. 3b. The dots show the elements of the unfolded tensor arranged as a matrix connected by a green line in the order that they are stored in memory. When $n = 1$, $\bar{\mathbf{Y}}_{(1)}$ is in column-major order; and when $n = 4$, $\bar{\mathbf{Y}}_{(4)}$ is in row-major order. For the interior modes, the data is a series of row-major subblocks. For $n = 2$, there are 4 subblocks of size 2×2. For $n = 3$, there are 2 subblocks of size 2×4. In local computations, each subblock can be processed separately using BLAS subroutines.

V. Parallel Kernels and Analysis

A. Parallel Cost Model and Collectives

To analyze our algorithms, we use the α-β-γ model of distributed-memory parallel computation, assuming the time to send a message of size W words between any two processors is $\alpha + W\beta$, where α is the latency cost and β is the per-word transfer cost, and γ is the time for one floating point operation (flop). For more discussion of the model and descriptions of efficient collectives, see [4], [19]. Let W be the data size and P be the number of processors, then the costs for relevant operations are summarized in Tab. I. For simplicity of presentation, we will ignore the flop cost of reduce and all-reduce, as they are typically dominated by the bandwidth costs.

Table I
Communication costs in α-β-γ model.

Send/Receive	$\alpha + \beta W$
All-gather	$\alpha \log P + \beta \frac{P-1}{P} W$
Reduce	$\alpha \log P + (\beta + \gamma) \frac{P-1}{P} W$
All-reduce	$2\alpha \log P + (2\beta + \gamma) \frac{P-1}{P} W$

B. Parallel TTM Computation

We consider the parallel algorithm for the TTM operation,

$$\mathcal{Z} = \mathcal{Y} \times_n \mathbf{V},$$

for a generic tensor $\mathcal{Y}$ of size $J_1 \times J_2 \times \cdots \times J_N$ and a matrix $\mathbf{V}$ of size $J_n \times K$. The result is a tensor $\mathcal{Z}$ of size $J_1 \times \cdots \times J_{n-1} \times K \times J_{n+1} \times \cdots \times J_N$. We can rewrite the operation in matricized form as

$$\mathbf{Z}_{(n)} = \mathbf{V}\mathbf{Y}_{(n)}.$$

The matrices are distributed as described in Sec. IV, and our parallel algorithm is presented in Alg. 3. We divide $\bar{\mathbf{V}}$, the local portion of $\mathbf{V}$, of size $K \times J_n/P_n$, into P_n block rows, i.e., $\bar{\mathbf{V}}^{[\ell]}$ denotes the ℓth block row of size $K/P_n \times J_n/P_n$. We compute the local product one block row at a time, i.e., locally compute

$$\mathcal{W} = \bar{\mathcal{Y}} \times_n \bar{\mathbf{V}}^{[\ell]} \quad \text{or} \quad \mathbf{W}_{(n)} = \bar{\mathbf{V}}^{[\ell]} \bar{\mathbf{Y}}_{(n)},$$

and sum the results across all processors in the same column to yield the result for the ℓth member. The local computation is also a mode-n TTM, which we implement using `dgemm` within BLAS while respecting the local layout of the unfolded tensor as described in Sec. IV-C.

The blocking ensures that the size of the intermediate products, which is $K/P_n \times \hat{J}_n/\hat{P}_n$, is never larger than the

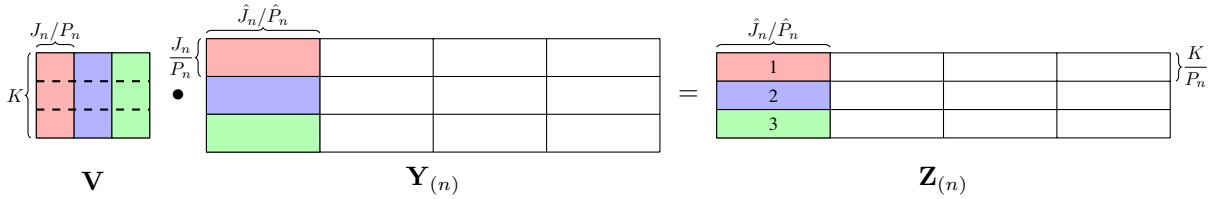


Figure 4. Parallel distributions of matrices involved in TTM operation $\mathcal{Z} = \mathcal{Y} \times_2 \mathbf{V}$ on a $2 \times 3 \times 2$ processor grid. The data owned by 3 of the 12 processors is color coded. The computation requires $P_2 = 3$ iterations, each using one block row of $\mathbf{V}$ and producing one (labeled) block of the output.

size of the local result tensor. We note that if $K < J_n/P_n$, then we can avoid the blocking strategy, performing a single local matrix multiplication followed by a single reduce-scatter, without the temporary memory exceeding the size of the local input tensor. This optimization reduces latency cost but does not affect computation or bandwidth costs; our analysis will assume the blocking strategy but our implementation exploits the non-blocked approach when possible.

Algorithm 3 Parallel TTM

1: **procedure** TTM($\mathcal{Y}$,$\mathbf{V}$,n)
2: `myProcID` $\leftarrow (p_1, p_2, \ldots, p_N)$
3: `myProcCol` $\leftarrow (p_1, \ldots, p_{n-1}, *, p_{n+1}, \ldots, p_N)$
4: **for** $\ell = 1, \ldots, P_n$ **do**
5: $\mathcal{W} \leftarrow \bar{\mathcal{Y}} \times_n \bar{\mathbf{V}}^{[\ell]}$
6: $\bar{\mathcal{Z}} \leftarrow$ REDUCE($\mathcal{W}$, `myProcCol`, ℓ) ▷ Root is $p_n{=}\ell$
7: **end for**
8: **return** $\mathcal{Z}$
9: **end procedure**

Fig. 4 illustrates the computation on a $2 \times 3 \times 2$ processor grid for $n = 2$. We implicitly treat the processor grid as $P_n \times \hat{P}_n = 3 \times 4$ and consider $\mathbf{Y}_{(n)}$ to be block distributed as described in Sec. IV-C. We color code the block column of $\mathbf{Y}_{(n)}$ owned by the first column in the processor grid; these three processors work together to compute the first block column of the result. Note that $\mathbf{V}$ is redundantly stored across every processor column as described in Sec. IV-B. The result tensor $\mathcal{Z}$ is partitioned in the same way as the input tensor $\mathcal{Y}$. Note that the blocks of the result are calculated one at a time, and numbered in the figure in the order that they are computed.

The cost and memory of the parallel TTM (Alg. 3) is

$$C_{\text{TTM}} = \underbrace{2\gamma\frac{JK}{P}}_{P_n\times \text{ line 5}} + \underbrace{\alpha P_n \log P_n + \beta(P_n - 1)\frac{\hat{J}_n K}{P}}_{P_n\times \text{ line 6}}, \text{ and}$$

$$M_{\text{TTM}} = \underbrace{J/P}_{\mathcal{Y}} + \underbrace{J_n K/P_n}_{\bar{\mathbf{V}}} + \underbrace{\hat{J}_n K/P}_{\bar{\mathcal{Z}}} + \underbrace{\hat{J}_n K/P}_{\mathcal{W}}.$$

The storage for $\mathcal{W}$ is temporary. If $P_n = 1$, then no parallel communication is required.

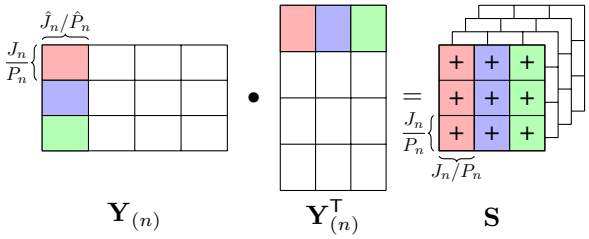


Figure 5. Parallel distribution of matrices involved in Gram operation $S = \mathbf{Y}_{(2)}\mathbf{Y}_{(2)}^T$ on a $2 \times 3 \times 2$ processor grid. The data owned by 3 of the 24 processors is color coded. Each processor column computes local matrix-vector products and the results are summed across each block row.

C. Parallel Gram Computation

Given mode n, we compute the Gram matrix $\mathbf{S} = \mathbf{Y}_{(n)}\mathbf{Y}_{(n)}^{\mathsf{T}}$ where $\mathcal{Y}$ is a tensor of size $J_1 \times \cdots \times J_N$. Here, we know $J_n = I_n$, but we ignore that fact in this discussion. The unfolded tensor $\mathbf{Y}_{(n)}$ is block distributed on the $P_n \times \hat{P}_n$ processor grid per Sec. IV-C. The result $\mathbf{S}$, of size $J_n \times J_n$, will be distributed with respect to mode n as described in Sec. IV-B. We ignore the fact that $\mathbf{S}$ is symmetric, storing both upper and lower triangles explicitly.

The data distribution for Gram is illustrated in Fig. 5, and the method is shown in Alg. 4. Each processor column owns a block column of $\mathbf{Y}_{(n)}$ and computes an intermediate matrix $\mathbf{V}$ of dimension $J_n \times J_n/P_n$. The matrix $\mathbf{V}$ is computed in row blocks of dimension $J_n/P_n \times J_n/P_n$, where each row block is the product of an unfolding of the local tensor with the unfolding of another processor's local tensor. The computation at line 5 is a local Gram computation that we perform using `dsyrk` within BLAS, though for interior modes it requires multiple subroutine calls to respect the local layout (as in the case of TTM). The computation at line 11 is a nonsymmetric analogue that we perform using `dgemm`. The $\mathbf{V}$ matrices are then summed across each processor row ($\hat{P}_n$ processors), using an all-reduce collective, so that the result (a block column of $\mathbf{S}$) is replicated across the processor row. Note that if $P_n = 1$, then the computation fully exploits symmetry, and the only communication is the all-reduce across P processors.

Algorithm 4 Parallel Gram

procedure GRAM($\mathcal{Y}$,n)
  myProcID $\leftarrow (p_1, p_2, \ldots, p_N)$
  myProcCol $\leftarrow (p_1, \ldots, p_{n-1}, *, p_{n+1}, \ldots, p_N)$
  myProcRow $\leftarrow (*, \ldots, *, p_k, *, \ldots, *)$
  $\mathbf{V}^{[p_n]} \leftarrow \bar{\mathbf{Y}}_{(n)} \bar{\mathbf{Y}}_{(n)}^{\mathsf{T}}$
  for $i = 1$ to $P_n - 1$ **do**
    $j \leftarrow (p_n - i) \mod P_n$
    $k \leftarrow (p_n + i) \mod P_n$
    Send $\mathcal{Y}$ to process $(p_1, \ldots, p_{n-1}, j, \ldots, p_N)$
    Receive $\mathcal{W}$ from process $(p_1, \ldots, p_{n-1}, k, \ldots, p_N)$
    $\mathbf{V}^{[k]} \leftarrow \bar{\mathbf{Y}}_{(n)} \mathbf{W}_{(n)}^{\mathsf{T}}$
  end for
  $\bar{\mathbf{S}} =$ All-Reduce($\mathbf{V}$, myProcRow)
  return $\bar{\mathbf{S}}$
end procedure

The cost of parallel Gram (Alg. 4) is

$$C_{\text{GRAM}} = \underbrace{\gamma 2 J_n J/P}_{\text{line 5 } + (P_n-1)\times \text{ line 11}} + \underbrace{2(P_n-1)(\alpha + \beta J/P)}_{(P_n-1)\times \text{lines 9 and 10}} + \underbrace{2\alpha \log \hat{P}_n + 2\beta(\hat{P}_n - 1)J_n^2/P}_{\text{line 13}}, \text{ and}$$

$$M_{\text{GRAM}} = \underbrace{J/P}_{\bar{\mathcal{Y}}} + \underbrace{J/P}_{\mathcal{W}} + \underbrace{J_n^2/P_n}_{\mathbf{V}} + \underbrace{J_n^2/P_n}_{\bar{\mathbf{S}}}.$$

The storage for $\mathcal{W}$ and $\mathbf{V}$ is temporary. Note that we report costs of our algorithm; up to a factor of two could be saved by exploiting symmetry of $\mathbf{S}$.

D. Parallel Eigenvectors Computation

Alg. 5 presents our method for computing the leading eigenvectors of the Gram matrix. After the Gram computation, the matrix $\mathbf{S}$ of size $I_n \times I_n$ is stored redundantly on every processor column in the $P_n \times \hat{P}_n$ processor grid as described in Sec. IV-B. We enforce the same distribution of the transpose of the output $I_n \times R_n$ eigenvector matrix $\mathbf{U}^{(n)}$, which implies a block row distribution of $\mathbf{U}^{(n)}$. Because we assume I_n is relatively small, e.g., $I_n \leq 2000$, our approach is essentially a sequential algorithm. We perform an all-gather across the processor fiber so that every processor owns the entire matrix $\mathbf{S}$. Every processor performs the local eigenvector computation redundantly using `dsyevx` within LAPACK and then extracts the appropriate subset of its local result to obtain the desired final distribution. The algorithm is presented in Alg. 5. While we assume R_n is an input to the algorithm, we can also choose R_n "on the fly" based on the desired error threshold for similar cost.

The cost and memory of the eigenvector computation (Alg. 5) is

$$C_{\text{EIG}} = \underbrace{\alpha \log P_n + \beta \frac{P_n - 1}{P_n} I_n}_{\text{line 4}} + \underbrace{\gamma \frac{10}{3} I_n^3}_{\text{line 5}}, \text{ and}$$

$$M_{\text{EIG}} = \underbrace{I_n^2/P_n}_{\bar{\mathbf{S}}} + \underbrace{I_n^2}_{\mathbf{S}} + \underbrace{I_n R_n}_{\mathbf{U}^{(n)}} + \underbrace{I_n R_n/P_n}_{\bar{\mathbf{U}}^{(n)}}.$$

Algorithm 5 Parallel Eigenvectors

procedure EIGENVECTORS($\bar{\mathbf{S}}$, R_n, n)
  myProcID $= (p_1, p_2, \ldots, p_N)$
  myProcCol $\leftarrow (p_1, \ldots, p_{n-1}, *, p_{n+1}, \ldots, p_N)$
  $\mathbf{S} =$ ALL-GATHER($\bar{\mathbf{S}}$, myProcCol)
  $\mathbf{U}^{(n)} =$ LOCAL-EIGENVECTORS($\mathbf{S}$, R_n)
  $\bar{\mathbf{U}}^{(n)} =$ ROW-SUBSET($\mathbf{U}^{(n)}$, P_n, p_n) ▷ Extract p_n-th block
  return $\bar{\mathbf{U}}^{(n)}$
end procedure

The memory for both "large" matrices ($\mathbf{S}$ and $\mathbf{U}^{(n)}$) is temporary and can overlap the local portions.

VI. Parallel Algorithm Analysis

A. ST-HOSVD

The initialization of the factor matrices in Alg. 2 is computed using ST-HOSVD in Alg. 1. The latency cost for ST-HOSVD is dominated by the TTMs, which is given by $\alpha \sum_{n=1}^{N} P_n \log P_n$. The remaining bandwidth and flop contribution of TTM within ST-HOSVD is

$$\frac{1}{P} \sum_{n=1}^{N} \left[(\beta(P_n - 1) + 2\gamma I_n) \prod_{k \leq n} R_k \prod_{k > n} I_k \right].$$

The total contribution of Gram to the cost of ST-HOSVD (aside from latency) is

$$\frac{1}{P} \sum_{n=1}^{N} \left[(2\gamma I_n + 2(P_n - 1)\beta) \prod_{k<n} R_k \prod_{k \geq n} I_k + 2\beta(\hat{P}_n - 1) I_n^2 \right]$$

The total contribution of calculating eigenvectors to the cost of ST-HOSVD (aside from latency) is

$$\sum_{n=1}^{N} \left[\beta \frac{P_n - 1}{P_n} I_n^2 + \gamma \frac{10}{3} I_n^3 \right].$$

The eigenvector cost is typically negligible compared to TTM and Gram. Compared to TTM, Gram has a factor of 2 on the bandwidth cost as well as a subtle change in limits of the product notation: the nth term in the Gram summation is a factor of I_n/R_n larger than the nth term in the TTM summation.

Note that the dominant expense in TTM and Gram depends on the ordering of modes; each iteration reduces the working data size. We can permute the modes in the main iteration to optimize the total cost (both computation and communication). We show some examples on the impact of these rearrangements in Sec. VIII-C.

In terms of memory, we can reuse the local variables from iteration to iteration and need not maintain temporaries. Therefore, the maximum storage *per processor* for ST-HOSVD is bounded above by

$$2I/P + \sum_{n=1}^{N} R_n I_n / P_n + \max_n I_n^2 + \max_n R_n I_n. \quad (2)$$

B. HOOI

The iterative improvement of the solution is computed using HOOI in Alg. 2. We derive the cost per outer iteration. The N inner iterations (computing each factor matrix update) can be done in any order, and this order does not affect the outer iteration cost. The latency cost of the TTMs dominates the cost of HOOI and each iteration is bounded above by $\alpha N \sum_{n=1}^{N} P_n \log P_n$. The bandwidth and computation cost from all $N(N-1)$ TTMs within one outer iteration of HOOI (ignoring the final TTM) is

$$\frac{N-1}{P} \sum_{n=1}^{N} (\beta(P_n - 1) + 2\gamma I_n) \prod_{k \leq n} R_k \prod_{k > n} I_k.$$

The total contribution of Gram (aside from latency) to the cost of one outer iteration of HOOI is

$$\frac{1}{P} \sum_{n=1}^{N} (2\beta(P_n - 1) + 2\gamma I_n) I_n \hat{R}_n + 2\beta I_n^2 (\hat{P}_n - 1).$$

The total contribution of the eigenvector calculations (aside from latency) to the cost of one outer iteration of HOOI is

$$\sum_{n=1}^{N} \left(\beta \frac{P_n - 1}{P_n} I_n + \gamma \frac{10}{3} I_n^3 \right).$$

This cost is typically dominated by the multiple-TTM computations in line 5, particularly so by the first TTM performed in each mode. As in ST-HOSVD, the multiple-TTM computations can be performed in any order, which can greatly affect run time. In fact, each of the N multiple-TTM computations can be ordered independently. We do not tune over these possibilities in this work.

The maximum memory is bounded above in the same way as for ST-HOSVD; see eq. (2).

VII. Application to Direct Numerical Simulation Data in Combustion Science

We demonstrate the utility of Tucker compression for two scientific data sets obtained by direct numerical simulation (DNS). A single simulation today can easily produce 100-1000 GB of data, and much more is expected in the future. Some attempts at compression using PCA and other methods have been made; see, e.g., [22]. Current data sizes are an obstacle for visualization and analysis because the data is difficult to transfer and requires high-end workstations or parallel clusters for computation. The goal of compression is to enable easier sharing of data and to facilitate analysis on reconstructed portions of the data. For instance, without reconstructing the entire data set, we can extract only the reconstruction of a single species, a few time steps, a coarser grid, a subset of the grid, or any combination of these. This enables the same data analysis to be performed on laptops.

A. Data Description

The DNS code used to produce this data is called S3D, a massively parallel compressible reacting flow solver developed at Sandia National Laboratories [5]. We work with the following multiway data sets:

- **HCCI-t:** This 4-way data tensor is of size $672 \times 672 \times 33 \times t$, requiring 71.6 GB storage for $t = 628$. It comes from the simulation of an autoignitive premixture of air and ethanol in Homogeneous Charge Compression Ignition (HCCI) mode [2]. The first two dimensions correspond to the 2D spatial grid, the third to the species, and the last to time.
- **TJ-A-t:** This 5-way data tensor is of size $300 \times 500 \times 240 \times 35 \times t$, requiring 122 GB storage for $t = 13$. It comes from a temporally-evolving planar slot jet flame with DME (dimethyl ether) as the fuel [3]. The first three dimensions correspond to the 3D spatial grid, the fourth to the species, and the last to time. This data has been significantly downsampled and so is less amenable to compression than the HCCI dataset.
- **TJ-B-t:** This 5-way data tensor is of size $460 \times 700 \times 360 \times 35 \times t$, requiring 512 GB storage for $t = 16$. It is the same as TJ-A, but higher spatial resolution.

Each data set is centered and scaled *for each species*. We compute the mean and standard deviation for each species slice, and then we transform the data by subtracting the mean and dividing the result by the standard deviation (unless it is less than machine precision, in which case the division in not performed). This normalizes the data so that we can roughly assume that each entry comes from a standard normal distribution.

B. Compression Rates

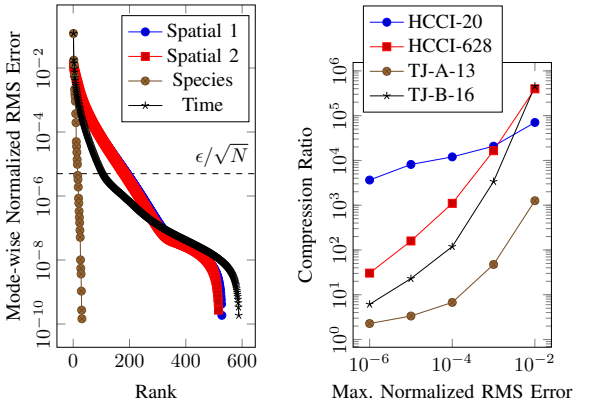
Let $\mathcal{X}$ be an N-way data tensor of size $I_1 \times \cdots \times I_N$. Let $\lambda_i^{(n)}$ denote the ith eigenvalue of the Gram matrix $\mathbf{X}_{(n)}\mathbf{X}_{(n)}^{\mathsf{T}}$ for $n = 1, \ldots, N$, corresponding to the square of the ith singular value of $\mathbf{X}_{(n)}$. Assume that the eigenvalues are in decreasing order, i.e., $\lambda_1^{(n)} \geq \lambda_2^{(n)} \geq \cdots \geq \lambda_{I_n}^{(n)}$. Let $\tilde{\mathcal{X}}$ be the reconstruction per eq. (1) using the T-HOSVD with reduced size $R_1 \times \cdots \times R_N$. It is well known [6], [21] that selecting R_n such that

$$\sum_{i=R_n+1}^{I_n} \lambda_i^{(n)} \leq \epsilon^2 \|\mathcal{X}\|^2 / N$$

for $n = 1, \ldots, N$ ensures that the T-HOSVD satisfies

$$\|\mathcal{X} - \tilde{\mathcal{X}}\|^2 \leq \sum_{n=1}^{N} \left(\sum_{i=R_n+1}^{I_n} \lambda_i^{(n)} \right) \leq \epsilon \|\mathcal{X}\|. \tag{3}$$

The normalized RMS error for HOOI initialized by ST-HOSVD is bounded above by the T-HOSVD error [6], [21]. In practice, computing $\{R_n\}$ given ϵ can be done within the



(a) Mode-wise contributions to the error bound for HCCI-628.

(b) Approximation error versus compression for different data sets.

Figure 6. Results on data sets from combustion science.

ST-HOSVD as specified in Alg. 1 (the T-HOSVD need not be computed).

Fig. 6a shows the normalized mode-wise RMS, i.e., $\left(\sum_{i=R_n+1}^{I_n} \lambda_i^{(n)}\right)^{1/2} / \|\mathcal{X}\|$ for each mode and value of R_n of HCCI-628. The rate of drop-off in the errors determines the compressibility of the data.

We show compression rates for all data sets in Fig. 6b, where the compression ratio is

$$C = \prod_{k=1}^{N} I_n \Big/ \left(\prod_{k=1}^{N} R_n + \sum_{k=1}^{N} I_n R_n \right).$$

The TJ-A-13 data set is the least compressible data set with C ranging from 2 at $\epsilon = 10^{-6}$ to 1200 for $\epsilon = 10^{-2}$. The HCCI-628 is much more compressible, with C ranging from 30 to 400000 for the same error range.

C. Reconstruction Error

Tab. II presents detailed compression results using $\epsilon = 10^{-5}$ to determine the compression ratios. We include the maximum absolute element error of the centered and scaled data. The HOOI iterations make little improvements on the ST-HOSVD initialization, so simply performing ST-HOSVD (with no HOOI iterations) is likely sufficient for this particular application area.

Table II
COMPRESSION AND MAXIMUM ABSOLUTE ELEMENTWISE ERROR FOR NORMALIZED RMS ERROR OF 1E-5.

Dataset	Reduced Size	Max. Elem. Error	Comp. Ratio
HCCI-1	(16, 16, 4, 1)	3.6e-5	573
HCCI-20	(20, 18, 6, 5)	2.0e-4	7083
HCCI-628	(192, 183, 16, 104)	1.2e-3	139
TJ-A-1	(257, 139, 186, 20, 1)	1.7e-3	9
TJ-A-13	(300, 209, 240, 25, 13)	3.2e-3	3

VIII. PERFORMANCE RESULTS

A. Experimental Platform

We run all experiments on Edison, a Cray XC30 supercomputer located at NERSC consisting of 5,576 dual-socket 12-core Intel "Ivy Bridge" (2.4 GHz) compute nodes. The peak flop rate of each core is 19.2 GFLOPS. Each node has 64 GB of memory. The nodes are connected by a Cray "Aries" interconnect with a dragonfly topology. We use Cray compilers and LibSci for BLAS and LAPACK subroutines.

B. Parameter Choice: Processor Grid Configuration

Our first microbenchmark demonstrates the effect of the processor grid on performance of the ST-HOSVD algorithm. Fig. 7a presents relative running times for a fixed problem size and number of processors, varying only the processor grid. We break down the running time across the three subroutines—Gram, Evecs, and TTM—which are each performed once in each mode. Thus, each bar has four blocks of each color, ordered from bottom to top so that the first subroutine computation corresponds to the bottommost block of that color. The dimensions are all equal to ensure the ordering of the modes does not impact the performance.

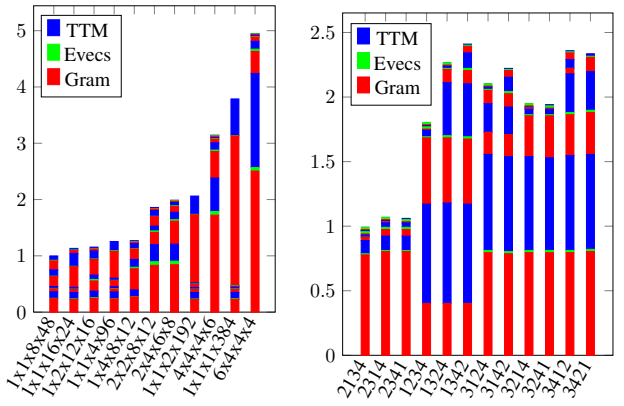
As shown in Sec. VI-A, the processor grid does not change the number of flops for each step in ST-HOSVD, but it does effect the performance of sequential linear algebra kernels due to its impact on the dimensions and layouts of the local matrices involved in the computations.

For most processor grids, the initial iteration consumes at least half of the overall running time. Gram dominates this initial iteration. As shown in the analysis of Sec. VI-A, the first Gram is more expensive than the first TTM by a factor of at least $I_1/R_1 = 4$ (in terms of both computation and communication). The cost of the Eigenvectors computation is negligible. The best processor grids have $P_1 = 1$, so that communication is minimized in the first iteration. In this case, the first Gram calculation only calls the all-reduce, which is of size I_1^2, and the first TTM involves no communication at all. We do not show results for processor grids with $P_1 > 6$, as they are more than 5 times the optimal running time.

The best processor grid for ST-HOSVD is not necessarily optimal for HOOI. While our limited tuning suggests that good choices for ST-HOSVD tend to be reasonable for HOOI and vice versa, an optimal overall processor grid choice for Alg. 2 depends on the number of iterations of HOOI.

C. Parameter Choice: Mode Ordering

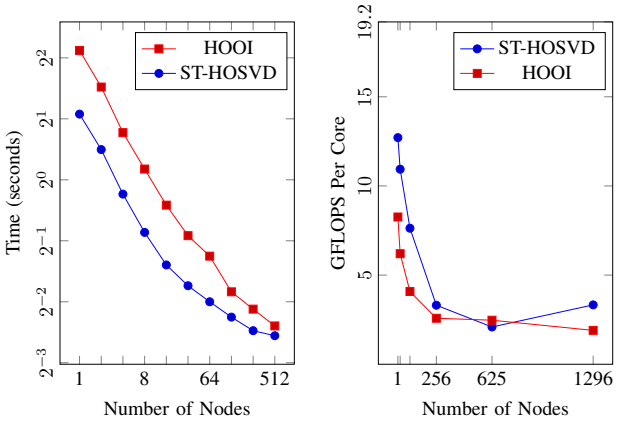
Our second microbenchmark demonstrates the effect of the mode ordering on performance of the ST-HOSVD algorithm. In Fig. 7b, we show relative run times for a fixed problem size and processor grid, varying only the order of modes in line 3 of Alg. 1. As in Fig. 7a, we break down



(a) Tensor of size 384^4 and reduced size of 96^4, varying processor grid.

(b) Vary mode order for $25\times250\times250\times250$ tensor with reduced size $10\times10\times100\times100$.

Figure 7. Relative run-time comparisons for different tuning options in ST-HOSVD.



(a) Strong scaling with using $24 \cdot 2^k$ processors, for $0 \le k \le 9$.

(b) Weak scaling for $(200k)^4$ tensor with reduced size $(20k)^4$, using $24 \cdot k^4$ processors (cores), for $1 \le k \le 6$.

Figure 8. Scaling performance of ST-HOSVD and one iteration of HOOI.

the run times across the three subroutines so that each bar comprises a block of each subroutine color for each mode.

For this problem, we use synthetic data. The full tensor dimensions are $25 \times 250 \times 250 \times 250$ which is formed from a tucker decomposition with core dimensions $10 \times 10 \times 100 \times 100$. We use a processor grid of dimensions $2 \times 2 \times 2 \times 2$. We vary both the tensor dimensions and the ratios of tensor to core dimensions to accentuate the effects of mode ordering. We use only 16 of 24 cores on one node in order to obtain a uniform processor grid and eliminate variability based on processor grid choice.

Most of the overall performance is determined by the choice of the first dimension, which is the smallest by a factor of 10. Starting with the first dimension means a cheaper first Gram but less reduction in computation and communication in subsequent iterations. Starting with the second dimension, which has the largest compression ratio, yields the greatest savings in subsequent iterations but incurs more overhead in the first iteration. Nevertheless, the optimal ordering starts with the second dimension. While the first Gram computation is more expensive than starting with the first dimension, the reduction in time for subsequent operations more than compensates.

We note that the authors of ST-HOSVD propose a heuristic ordering (in the case of sequential computation) that greedily chooses the mode that minimizes the number of flops in the current iteration [21]. While that heuristic is not optimal in this case, we see no simple alternative scheme that is always optimal. Another reasonable heuristic is to greedily choose the mode that maximizes the compression ratio I_n/R_n.

D. Strong Scaling

To test the parallel scaling of our algorithm, we fix a particular problem and increase the number of processors we use to compute the Tucker decomposition. Fig. 8a reports the running time of ST-HOSVD and one iteration of HOOI across $24 \cdot 2^k$ processors, for $0 \le k \le 9$.

For the experiment, we use the TJ-A-1 data set, which is a 4-way tensor representing one time step of a 3D combustion simulation (the fourth mode corresponds to species) with dimensions $500 \times 300 \times 240 \times 35$ We compress it to a core tensor of dimension $42 \times 115 \times 81 \times 10$ (yielding error of 10^{-3} and a compression ratio of 284). For each k, we tune the processor grid over three or four possibilities (chosen heuristically) for each algorithm and report the minimum running time.

On one node, the data set requires about 1/7th of the available memory. ST-HOSVD and one step of HOOI runs in about 6 seconds, achieving 83% and 40% of peak performance, respectively. On 512 nodes, the same computation requires 0.36 seconds, achieving aggregate performance of 4.7 and 4.2 TFLOPS, respectively. At this scale, the local tensor data is about 750 KB per core. We note that we continue to decrease running time even at high concurrency, though we are far from peak performance, as the interprocessor communication and small matrix dimensions within local computation kernels both degrade performance.

E. Weak Scaling

Fig. 8b shows a weak scaling experiment, reporting performance per core for both ST-HOSVD and one iteration of HOOI. In this experiment, we fix the amount of data per processor and increase the number of processors and tensor dimensions simultaneously.

We use $24 \cdot k^4$ processors and set the tensor dimensions to be $(200k)^4$ with cores of dimension $(20k)^4$, for $1 \le k \le 6$. The plot reports the best performance for each algorithm over three different processor grids: $1 \times 1 \times 4k^2 \times 6k^2$, $k \times$

$k \times 4k \times 6k$, and $k \times 2k \times 3k \times 4k$. The size of the data sets ranges from about 12 GB for $k = 1$ up to 15 TB for $k = 6$. On one node ($k = 1$), ST-HOSVD and HOOI achieve 66% and 43% of peak performance, respectively. For 1296 nodes ($k = 6$), the algorithms achieves 17% and 12% of peak, respectively, or up to 104 TFLOPS in aggregate. The time required to process the 15 TB data set (performing ST-HOSVD and HOOI on data in memory) is 70 seconds.

The main reason that we see degradation in performance as we scale up to high processor counts is that it becomes harder to navigate the tradeoffs among optimization of the processor grid for the computations in different modes.

IX. Conclusion

Our parallel Tucker decomposition enables compression of massive data sets that do not fit into memory on a single machine. We show that these data sets can be processed in reasonable time and yield excellent compression rates. The implementation performs well (near peak performance) at low core counts, and it scales (offers reduced run times) up to high core counts. In order to achieve even better parallel scaling, we see several avenues for performance improvement such as using multi-threaded BLAS for all local computations or adapting recent work in optimizing multi-threaded TTM computations [15]. Additionally, we can overlap communication and computation and fully exploit the symmetry in the Gram computation. For achieving approximation errors near the square root of machine precision (or smaller), we need to consider a numerical improvement to our algorithm, directly computing the singular values. Improving the numerical stability of our algorithm will not drastically hurt our overall performance; because $\mathbf{Y}_{(n)}^{\mathsf{T}}$ is typically very tall and skinny, we can compute the SVD using a QR decomposition as a preprocessing step at roughly twice the cost of our current approach.

Acknowledgment

We thank Hemanth Kolla and Ankit Bhagatwala for providing the combustion application data. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This material is based upon work supported by the Sandia Truman Postdoctoral Fellowship and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE–AC04–94AL85000.

References

[1] R. Ballester-Ripoll and R. Pajarola, *Lossy volume compression using Tucker truncation and thresholding*, Vis Comput, (2015), doi:10.1007/s00371-015-1130-y.

[2] A. Bhagatwala, J. H. Chen, and T. Lu, *Direct numerical simulations of HCCI/SACI with ethanol*, Combustion and Flame, 161 (2014), pp. 1826–1841, doi:10.1016/j.combustflame.2013.12.027.

[3] A. Bhagatwala, Z. Luo, H. Shen, J. A. Sutton, T. Lu, and J. H. Chen, *Numerical and experimental investigation of turbulent DME jet flames*, Proc. Combustion Institute, 35 (2015), pp. 1157–1166, doi:10.1016/j.proci.2014.05.147.

[4] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, *Collective communication: theory, practice, and experience*, Concurrency and Computation: Practice and Experience, 19 (2007), pp. 1749–1783, doi:10.1002/cpe.1206.

[5] J. H. Chen et al., *Terascale direct numerical simulations of turbulent combustion using S3D*, Computational Science & Discovery, 2 (2009), 015001, doi:10.1088/1749-4699/2/1/015001.

[6] L. De Lathauwer, B. De Moor, and J. Vandewalle, *A multilinear singular value decomposition*, SIAM J. Matrix Analysis and Applications, 21 (2000), pp. 1253–1278, doi:10.1137/S0895479896305696.

[7] L. De Lathauwer, B. De Moor, and J. Vandewalle, *On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors*, SIAM J. Matrix Analysis and Applications, 21 (2000), pp. 1324–1342, doi:10.1137/S0895479898346995.

[8] S. Etter, *Parallel ALS algorithm for the hierarchical Tucker representation*, Report 2015-25, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2015.

[9] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, *GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries*, in KDD'12, 2012, pp. 316–324, doi:10.1145/2339530.2339583.

[10] A. Karami, M. Yazdi, and G. Mercier, *Compression of hyperspectral images using discerete wavelet transform and tucker decomposition*, IEEE J. Selected Topics in Applied Earth Observations and Remote Sensing, 5 (2012), pp. 444–450, doi:10.1109/JSTARS.2012.2189200.

[11] L. Karlsson, D. Kressner, and A. Uschmajew, *Parallel algorithms for tensor completion in the CP format*, report, Universität Bonn, 2014.

[12] O. Kaya and B. Uçar, *Scalable sparse tensor decompositions in distributed memory systems*, Report RR-8722, INRIA, Research Centre Grenoble, Rhône-Alpes, France, 2015.

[13] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Review, 51 (2009), pp. 455–500, doi:10.1137/07070111X.

[14] P. M. Kroonenberg and J. De Leeuw, *Principal component analysis of three-mode data by means of alternating least squares algorithms*, Psychometrika, 45 (1980), pp. 69–97, doi:10.1007/BF02293599.

[15] J. Li, C. Battaglino, I. Perros, J. Sun, and R. Vuduc, *An input-adaptive and in-place approach to dense tensor-times-matrix multiply*, in SC'15, 2015.

[16] A. Phan and A. Cichocki, *PARAFAC algorithms for large-scale problems*, Neurocomputing, 74 (2011), pp. 1970–1984, doi:10.1016/j.neucom.2010.06.030.

[17] M. Schatz, *Distributed Tensor Computations: Formalizing Distributions, Redistributions, and Algorithm Derivations*, PhD thesis, University of Texas, Austin, 2015.

[18] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, *SPLATT: Efficient and parallel sparse tensor-matrix multiplication*, in IPDPS'15, 2015, pp. 61–70, doi:10.1109/ipdps.2015.27.

[19] R. Thakur, R. Rabenseifner, and W. Gropp, *Optimization of collective communication operations in MPICH*, Intl. J. High Performance Computing Applications, 19 (2005), pp. 49–66, doi:10.1177/1094342005051521.

[20] L. R. Tucker, *Some mathematical notes on three-mode factor analysis*, Psychometrika, 31 (1966), pp. 279–311, doi:10.1007/BF02289464.

[21] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, *A new truncation strategy for the higher-order singular value decomposition*, SIAM J. Scientific Computing, 34 (2012), pp. A1027–A1052, doi:10.1137/110836067.

[22] Y. Yang, S. B. Pope, and J. H. Chen, *Empirical low-dimensional manifolds in composition space*, Combustion and Flame, 160 (2013), pp. 1967–1980, doi:10.1016/j.combustflame.2013.04.006.

[23] G. Zhou, A. Cichocki, and S. Xie, *Decomposition of big tensors with low multilinear rank*, 2014, arXiv:1412.1885.